RECOMMENDATION FOR CHANGE IN GAME ENGINE: FIT2099 SSB ASSIGNMENT 3

GROUP NAME	**JAVAPLUSPLUS**	
GROUP MEMBERS	MOHAMED SHAKEEL MOHAMED RAFI	28021452
	MATTI HADDAD	29708966
	KERRY YUE SONG ZHENG	28794346

Nonplayer Actors not taking direct advantage of Actions

The engine does not allow nonplayer actors to directly perform actions, such as Give and Attack. Rather it executes their own act() method which decides how the actor behaves. This creates duplicated code for nonplayer actors who share common behaviour such as Give and Attack. One way to fix this is to have the engine treat each actor same as the player. Each Actor has a set of actions and each actor set preference for each action.

for example: Dumbledore has three actions

1. Give – preference 1
2. Attack – preference 2
3. Move – preference 3

The engine, should check the canDo() method of each Action for Dumbledore, then it should execute the highest preference one assuming the action can be done by Dumbledore.

Items Doing too much

With current implementation, items that can be picked by actors, have their own affordance such as Take and Leave, which requires the developer to maintain and keep track of affordances in two areas, Actors and Nonactors(items). Items do not require these responsibilities- they are doing too much. One way to break this relationship, is to make actors decide whether they can perform/canDo() this action or not such as Leave and Take. This leads to the conclusion that items should have a class of their own that implements EntityIterface, by doing so, items affordances can be transfer to actors, this will result in less dependencies and association.

Abstraction

Excessive use of public methods rather than other access modifier such as private and protected, doing so might introduce new dependences, increase complexity and therefore increase the maintenance as the software develops over time.

This is especially true when developers look for quick and dirty way to fix bugs or to implement new features, this may also lead to violation of 'Separation of concerns' principle. Thus, such design may result in the creation of similar or duplicated methods to overcome time constraint.

Nonmeaningful variables

The use of nonmeaningful variables such as s, l, e makes it harder to understand the underlying code and may lead to reduced productivity. Developer constantly need to refer to the declaration of these variable to comprehend the function of a module.

Lazy classes

Some classes such as MapRender MessageRendered, Direction and SimulationController barely have one method and they are not doing much. These classes cost to maintain and understand. Therefore, it's better to eliminate them and merge their functionality into a single class (MapRender MessageRendered into Renderer) or the most class that is shares the similar responsibility. Thus, reducing the cost of maintenance.

Encapsulation

The getters and setters of some classes especially in affordance and location allows for clients to changes the attributes of these classes rendering the use of making attributes private. A solution to this would be to create defensive copies for such methods. Although this would increase the time complexity and may slightly increase the overall running speed of the game engine, it would prevent privacy leaks and would not result in a loss of encapsulation

Literals

The excessive use of literals especially in Grid makes it hard to understand and difficult to maintain should the requirements for the world change and there needs to be a different set of coordinates and movements in the game. A solution to this would be to refactor the code so that the literals are made as attributes that can be easily modified instead of meaningless numbers put into methods, so that it is more readable and can be easily changed.